Nima Dastmalchi (505320372)
CS 131 Homework 7

Problem 1
Below pseudocode describes how short-circuiting of the if-statements works.

```
# First if statement:
if (e())
    do_something()
else if (f())
    if (g())
        do_something()
else if (h())
    do_something()

# Second if statement:
if (e())
    if (f())
        do_something()
else if (g())
    if (h())
        do_something()

# Third if statement:
if (e())
    if (f())
        do_something()
    else if (g())
        do_something()
```

Problem 2 Part a
Interface A has no supertypes
Interface B has supertype A
Interface C has supertype A
Class D has supertypes B, C, A
Class E has supertypes C, A
Class F has supertypes D, B, C, A
Class G has supertypes B, A

Problem 2 Part b
All objects of type B or that have a supertype B can be passed to function foo (assuming that this language passes parameters by object reference or reference).

So, objects of classes D, F, and G can be passed to function foo.

Problem 2 Part c
No, we cannot call bar(a). This is because objects of classes that implement A do not necessary also implement C. Thus, the variable 'a' could be pointing to an object that does not implement the interface C and cannot be casted to a variable of type C.

Problem 3
Inheritance is when a subclass extends a superclass. In this case, the subclass gets access to both the interface and the implementation of the superclass.

Subtype polymorphism is the ability to pass in an object of the subtype (type of the subclass) to any procedure/function/piece of code that is expecting a supertype (type of the superclass of the subclass).

Dynamic dispatch is the process of determining the method to call on an object during runtime. Dynamic dispatch is used in both statically typed and dynamically typed languages. In statically typed languages, we may have an object of a subtype referred to by a supertype variable. Since the method can be overridden by the subclass, we must determine which method implementation to run during runtime. On the other hand, the methods of an object in a dynamically typed language can change at any time, so we have to check at runtime whether or not the method exists.

Problem 4
By definition, in subtype polymorphism, we have a piece of code that expects a supertype variable that potentially refers to a subtype object. However, variables in dynamically typed languages do not have types, so that variable can point/refer to any object.

We still need dynamic dispatch in dynamically typed languages because we cannot know, at compile-time (or before runtime), what methods an object holds. Below code shows an example of why dynamic dispatch is necessary.

```
obj = { method1: () => console.log("hello world") }
if (unknownVariableAtTranspileTime) {
    obj.method1 = () => console.log("new function")
}
obj.method1()
```

Problem 5

1. **Dependency Inversion Principle**. We should create a Charger interface that is implemented by the SuperCharger class. The ElectricVehicle class should operate on all objects of the Charger interface for flexibility. (note: since C++ does not support interfaces, we make Charger interface an abstract class instead).

```
class Charger {
public:
    virtual void get_power() = 0;
    virtual double get_max_amps() const = 0;
    virtual double check_price_per_kwh() const = 0;
};

class SuperCharger : public Charger {
public:
    void get_power() { ... }
    double get_max_amps() const { ... }
    double check_price_per_kwh() const { ... }
};

class ElectricVehicle {
public:
    void charge(Charger& sc) { ... }
};
```

2. **Single Responsibility Principle**. The SuperCharger class has too much responsibility. The SuperCharger class implements the check_price_per_kwh method. The result of this method may result in external events not related to the Charger (e.g., current price of electricity). Thus, we could implement a ElectricPriceCalculator class that is passed into the SuperCharger class.

```
3. class ElectrityPriceCalculator {
4.     double get_price() const { ... }
5. };
6.
7. class SuperCharger {
8. public:
9.     void get_power() { ... }
10.    double get_max_amps() const { ... }
11.    double check_price_per_kwh(ElectrityPriceCalculator&) const { ... }
12. };
```

Problem 7 Part a

```python
class Node:
    def __init__(self, val):
        self.value = val
        self.next = None

class HashTable:
    def __init__(self, buckets):
        self.array = [None] * buckets
    def insert(self, val):
        bucket = hash(val) % len(self.array)
        tmp_head = Node(val)
        tmp_head.next = self.array[bucket]
        self.array[bucket] = tmp_head
    def __iter__(self):
        return HashTableIterator(self)

class HashTableIterator:
    def __init__(self, hashTable : HashTable):
        self.hashTable = hashTable
        self.generator = hashTableGenerator(self.hashTable)

    def __next__(self):
        return next(self.generator)

def hashTableGenerator(hashTable : HashTable):
    for current in hashTable.array:
        while current != None:
            yield current.value
            current = current.next

ht = HashTable(10)
ht.insert(10)
ht.insert(20)
ht.insert(30)
for x in ht:
    print(x)

# Output:
# 30
# 20
# 10
```

Problem 7 Part b

Similar approach to part a, but instead of using a generator function, we keep track of the position index in an array containing elements of the hash table.

```
class Node:
    def __init__(self, val):
        self.value = val
        self.next = None

class HashTable:
    def __init__(self, buckets):
        self.array = [None] * buckets
    def insert(self, val):
        bucket = hash(val) % len(self.array)
        tmp_head = Node(val)
        tmp_head.next = self.array[bucket]
        self.array[bucket] = tmp_head
    def __iter__(self):
        return HashTableIterator(self.array)

class HashTableIterator:
    def __init__(self, array):
        self.flat_array = []
        # create a flat array that contains all elements of the hash table so
        # we can iterate through all elements of the hash table with 1 pos index
        for current in array:
            while current is not None:
                self.flat_array.append(current)
                current = current.next
        # save the current position in the flat_array
        self.pos = 0
    def __next__(self):
        if self.pos >= len(self.flat_array):
            raise StopIteration()
        ret = self.flat_array[self.pos]
        self.pos += 1
        return ret.value

ht = HashTable(10)
ht.insert(10)
ht.insert(20)
ht.insert(30)
for x in ht:
    print(x)
# Output:
# 30
# 20
# 10
```

Problem 7 Part c

The for loop is at the bottom of the answers to the previous 2 questions:

```
ht = HashTable(10)
ht.insert(10)
ht.insert(20)
ht.insert(30)
for x in ht:
    print(x)
```

Note that both implementations will output:

```
# Output:
# 30
# 20
# 10
```

Problem 7 Part d

Manual loop:

```
ht = HashTable(10)
ht.insert(10)
ht.insert(20)
ht.insert(30)

iterator = iter(ht)
try:
    while True:
        print(next(iterator))
except StopIteration:
    pass
```

Problem 7 Part e

```
class HashTable:
    def __init__(self, buckets):
        self.array = [None] * buckets
    def insert(self, val):
        bucket = hash(val) % len(self.array)
        tmp_head = Node(val)
        tmp_head.next = self.array[bucket]
        self.array[bucket] = tmp_head
    def forEach(self, f):
        for current in self.array:
            while current is not None:
                f(current.value)
                current = current.next

ht = HashTable(10)
ht.insert(10)
ht.insert(20)
```

```
ht.insert(30)

ht.forEach(lambda x: print(x))

# Output:
# 30
# 20
# 10
```